P3 Fast API

Fast API

https://www.youtube.com/watch?v=tLKKmouUams

CRASH COURSE VIDEO
https://www.youtube.com/watch?v=tLKKmouUams

TEXT DOCS
https://www.tutorialspoint.com/fastapi/index.htm

https://fastapi.tiangolo.com/tutorial/body-multiple-params/

Basics & Setup

Running Webservers w/ Uvicorn

NOT A FULL GUIDE
Next lesson will cover that - this just gives you pre-requisite knowledge

Purpose of Uvicorn

Uvicorn will take our endpoints and run them on a local webserver

- By running our them on a webserver, Uvivorn allows you to make contact with the endpoints using endpoint calls
- Without Uvicorn or an alternative, our endpoints would not be reachable and we could not test them

Setting up a Dummy Endpoint

To demonstrate how to run Uvicorn, we're setting up a very simple GET endpoint that returns static data
The GET endpoint syntax will be the focus of a future lesson

main.py

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def index():
    return {"detail": 1000}
```

How to use Uvicorn

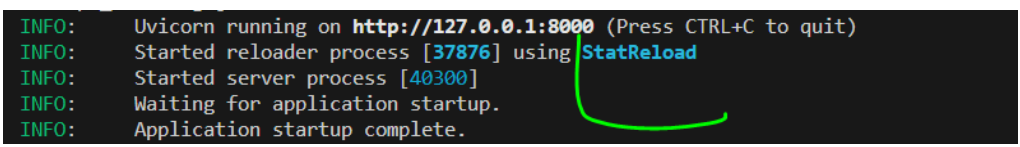
ASSUMPTION:
Custom endpoints have been set up already

1. Run the webserver
   Open up a terminal in the directory where your central python file is to run this command

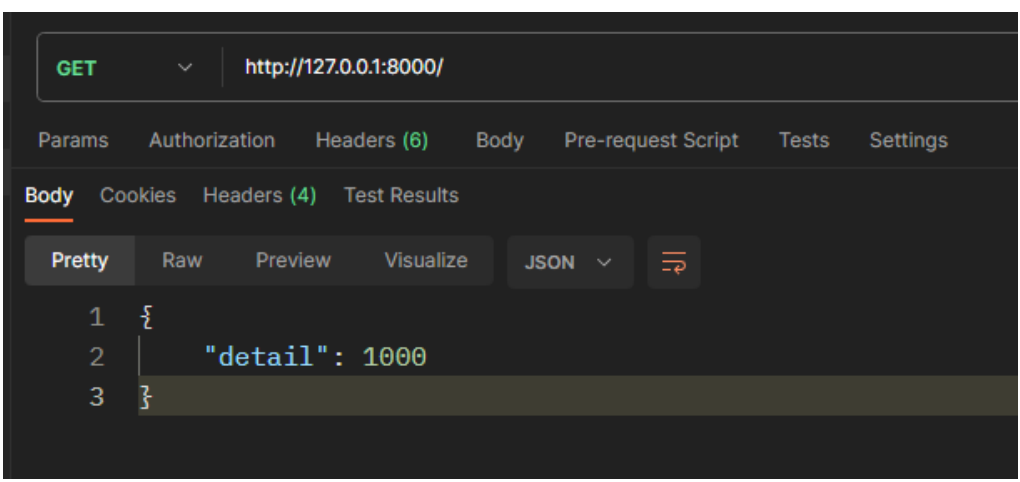
   - main references main.py (name may change)
   - app references what we set `FastAPI()` equal to on our central python file (name may change)

```
uvicorn main:app --reload
```

1. Read the terminal and find the address where your local webserver is hosting your endpoints


INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [37876] using StatReload
INFO: Started server process [40300]
INFO: Waiting for application startup.
INFO: Application startup complete.

1. Make request to the address found in step 2- in this case it's `http://127.0.0.1:8000/`



How to Set up a FastAPI Project from Scratch (MAIN)

PRE-REQUISITE

Make sure you have Python and PIP installed on your machine
(See other files for an explanation on how to do that)

What are we Setting Up?

We're setting up a Python server that allows you to specify endpoints

- They will be reachable once you broadcast the project with uvicorn
- All dependencies are installed onto a virtual environment, not globally (professional)

Step by Step Guide

1. Create a folder and initialize the repo

```
git init
```

1. Set up a virtual environment- command creates a .venv folder

```
py -m venv .venv
```

1. Activate virtual env
   We need to install dependencies on the virtual environment, not globally

```
source .venv/Scripts/activate   (for Linux users)
.venv\Scripts\activate     (for Windows users)
```

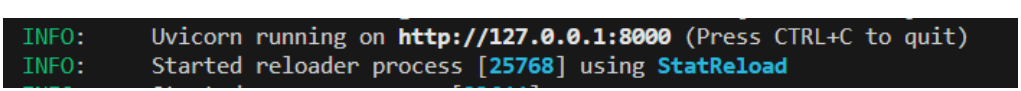
1. Install dependencies with pip or poetry

```
pip install fastapi  (mandatory)
pip install uvicorn  (mandatory)
pip install requests (probably needed)

poetry add fastapi  (mandatory)
poetry add uvicorn  (mandatory)
poetry add requests
```

1. Activate uvicorn to broadcast your python server
   The command below assumes `main.py` is the name of your project's central file, and that it's on the project's root level

```
uvicorn main:app --reload
```



1. You can now make requests to endpoints on your server

`deactivate` (use this when you want to deactivate the virtual env)

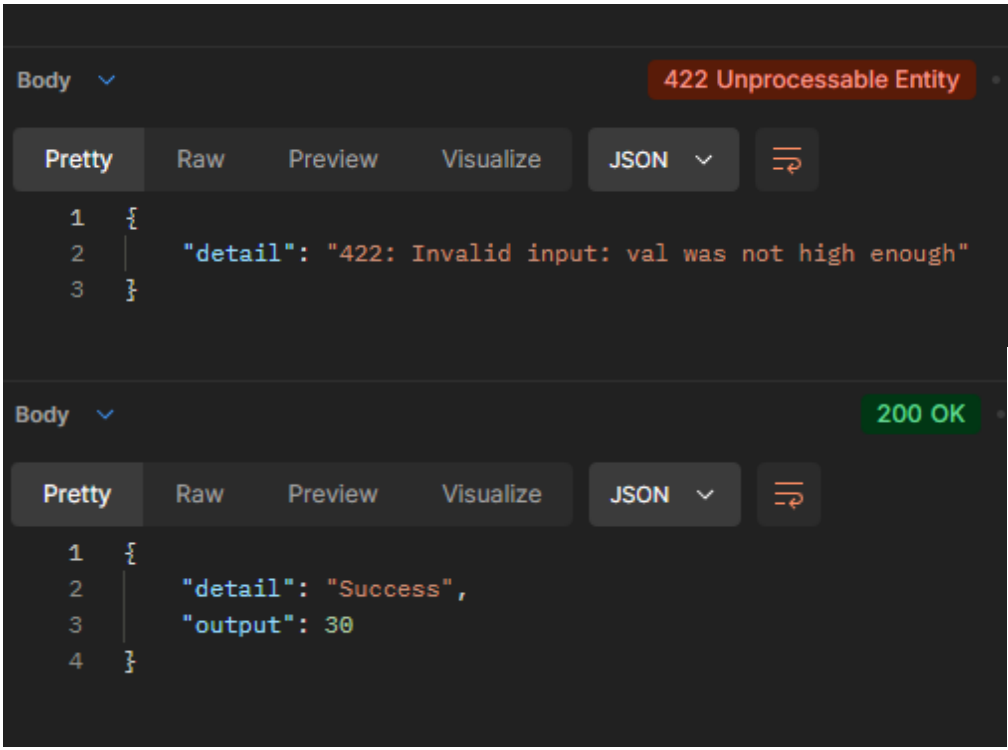
Error Handling in FastAPI

FastAPI is smart enough to generate a response object whenever an exception is raised
You can easily set the status code and an error message

```
@router.get(path="/callback/{val}")
async def get_callback(val: int):
    try:
        if val < 10:
            raise HTTPException(status_code=422, detail="In
        return {"detail": "Success", "output": val}

    except Exception as err:
        status_num = err.status_code or 500
        error_msg = str(err) or "An error has occurred in t
        raise HTTPException(status_code=status_num, detail=
```

Request made to: `http://localhost:8090/v1/callback/5` Request made to: `http://localhost:8090/v1/callback/30`


Body
422 Unprocessable Entity
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "detail": "422: Invalid input: val was not high enough"
3 }
Body
200 OK
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "detail": "Success",
3 "output": 30
4 }

Extract Data fr/ Endpoint Calls

Path Parameters

Our endpoints are hosted on a webserver at this address in our examples: `http://127.0.0.1:8000/`

Static Path Parameters

Create endpoint that fires when making a call to `/purchases`

```
@app.get("/purchases/")
def index():
    return {"cost": 18}
```

Create endpoint that fires when making a call to `/sales/discover`

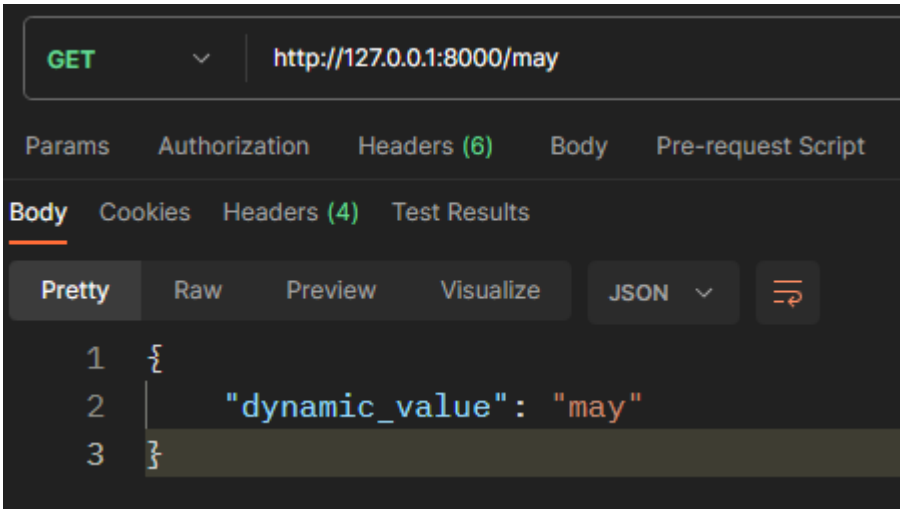
```
@app.get("/sales/discover/")
def index():
    return {"cost": 18}
```

Dynamic Path Parameters

Create endpoint that fires when making a call to `/{literally_anything}`
And use that dynamic path parameter in the endpoint by returning it

```
@app.get("/{name}")
def index(name):
    return {"dynamic_value": name} # access dynamic path pa
```

Made request to `http://127.0.0.1:8000/may`

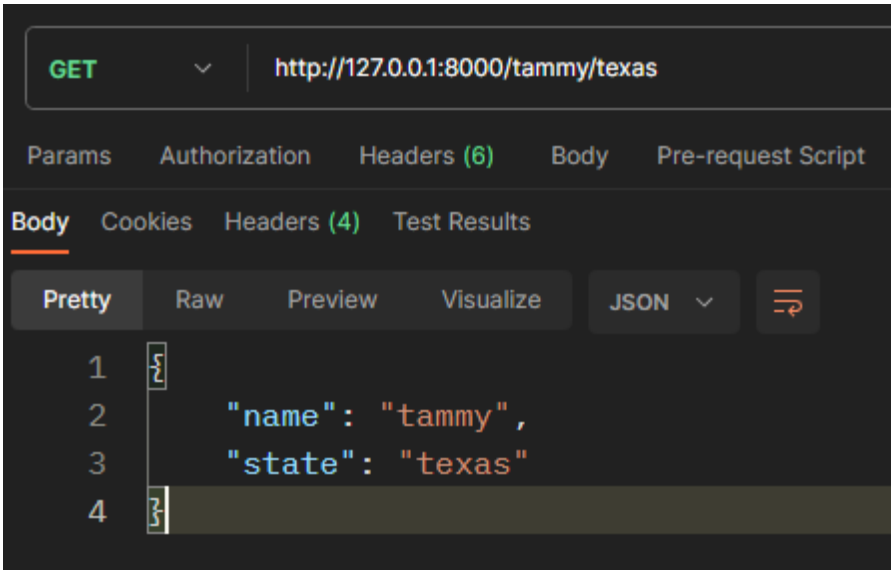


Multiple Dynamic Path Parameters

Create an endpoint that fires when making a call to `/{something}/{something}`
And use all dynamic path parameters in the endpoint by returning them

```
@app.get("/{name}/{state}")
def index(name, state):
    return {"name": name, "state": state}
```

Made request to `http://127.0.0.1:8000/tammy/texas`



Unspecified Number of Path Params

Use `{some_name:path}` to capture the entire path in a string, including forward slashes

```
@router.get("/schemas/{namespace:path}")
async def fetch_schemas(namespace: str):
    return { "detail": namespace }
# If we use the following path: http://localhost:8060/v1/sc
# We get this as a return: { "detail": "chain/document_mana
```

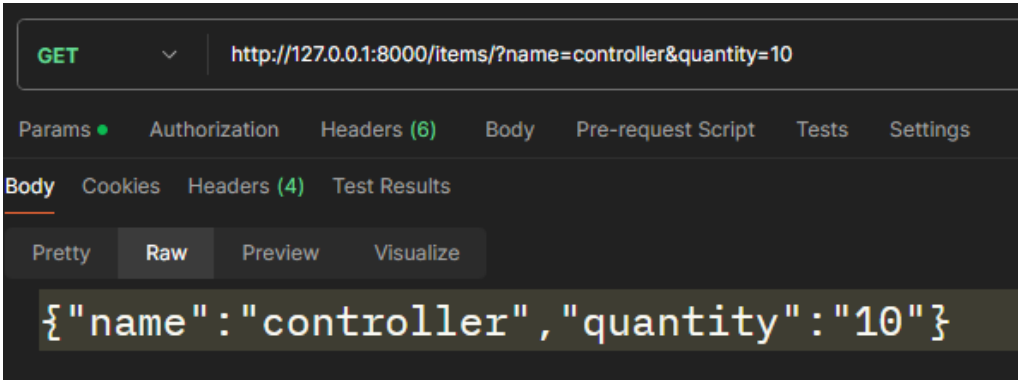
Query Parameters: Basics

Access Query Params

You can access query params in the arguments list of the function attached to your endpoint

```
@app.get("/items/")
```

```python
def read_items(name, quantity):
    return {"name": name, "quantity": quantity}
```

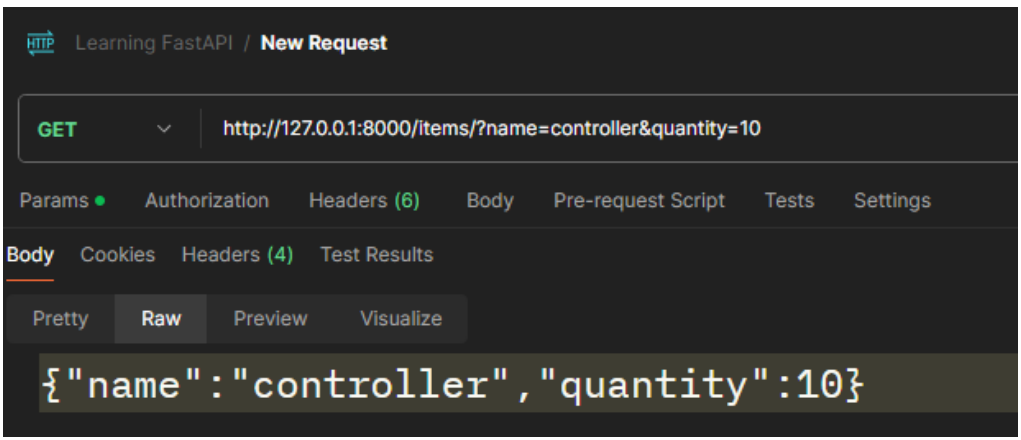


Type Conversion

You can declare query parameters with a type
Doing so will turn the values passed using the query string into a type

EXAMPLE:
Notice how quantity of 10 is returned as a number in the example below, but as a string in the previous one

```python
@app.get("/items/")
def read_items(name: str, quantity: int):
    return {"name": name, "quantity": quantity}
```



Type Conversion for Validation

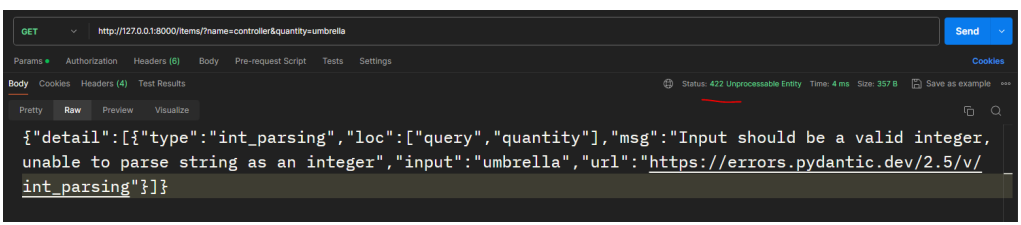
An exception will be raised raised if...

- If a certain query param is defined using a type
- You try to pass it a value with a type that doesn't match

EXAMPLE:
Pass the string "umbrella" to a query param whose type has been defined as an integer

```python
@app.get("/items/")
def read_items(name: str, quantity: int): # quantity query
    return {"name": name, "quantity": quantity}
```

Error code 422 is raised and we get an error response



List of Common Types

One item of type...

Lists of items with the same value type

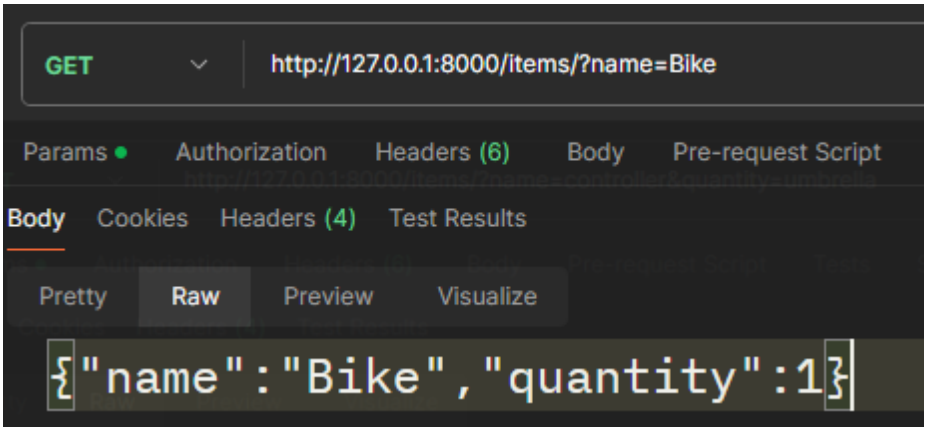
Set of items with the same value type

Query Parameters: Advanced

Default Values

We set the default value of quantity to 1
Now when we do not send a quantity query param, the value defaults to 1

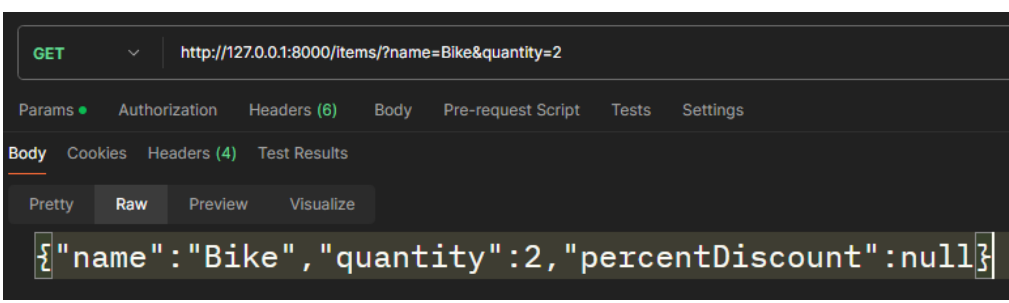
```python
@app.get("/items/")
def read_items(name: str, quantity: int = 1):
    return {"name": name, "quantity": quantity}
```



Optional Params

You can make query parameters optional by setting their default value to None

```python
@app.get("/items/")
def read_items(name: str, quantity: int, percentDiscount: i
    return {"name": name, "quantity": quantity, "percentDis
```



< Send JSON Object through Query Params >

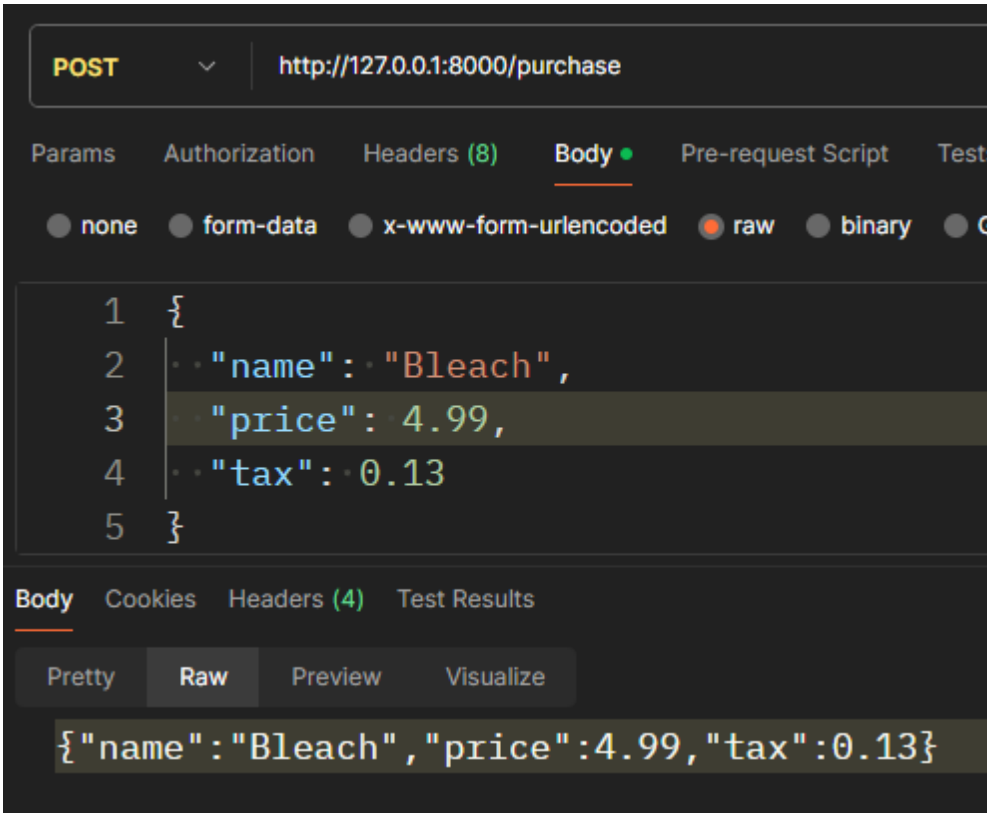
Will prob come up eventually

Request Body

Pass Data using Request Body

1. Define a schema for the request body using pydantic
2. Pass JSON through the request body when you reach out to your endpoint

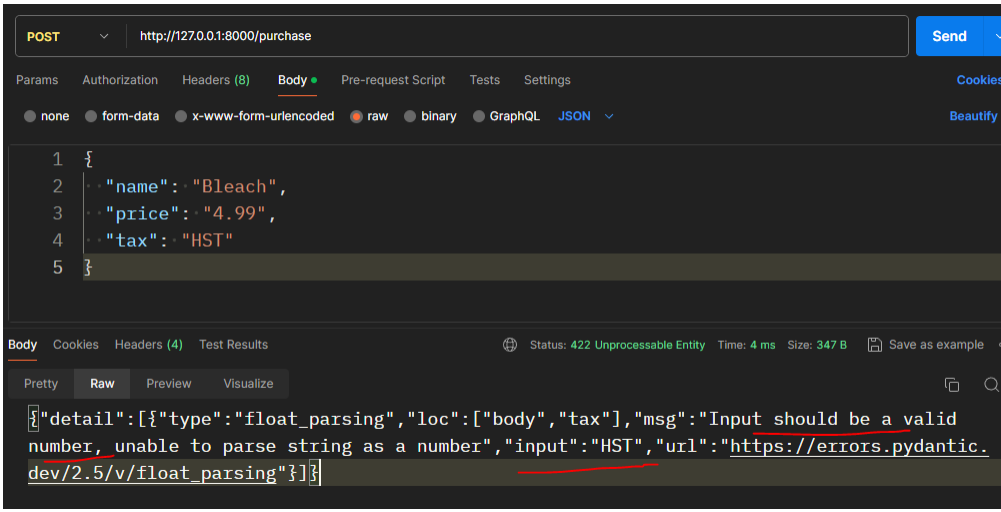
```python
from fastapi import FastAPI
from typing import Optional
from pydantic import BaseModel
app = FastAPI()
# Define schema for the request body
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.post("/purchase")
def read_items(item: Item):
    request_body = item.model_dump() # turn data into a Pyt
    return request_body
```



Pydantic helps with Validation

1. The schema will do its best to convert types
   Even when request body data and the schema type aren't an exact match
   - In the below example, we pass a price of "4.99" as a string, not a number
   - That's okay since it can be converted to a float easily thanks to pydantic

4. An exception gets raised if type conversion is not enough to make the request body data match the schema
   - In the below example, we pass a value of "HST" for tax, which is supposed to be a float
   - The string HST cannot be converted to a float of any kind, which results in error code 422 (exception raised)



Use Path Params + Query Params + Request Body

https://fastapi.tiangolo.com/tutorial/body/#request-body-path-query-parameters

It can get confusing when you use the following all at once: dynamic path parameters + query params + request body
They all appear in the endpoint's function argument list

Dynamic Path Param

```
@app.get("/{name}")
def index(name):
    return {"dynamic_value": name} # access dynamic path pa
```

Query Param

```
@app.get("/items/")
def read_items(name):
    return {"name": name}
```

Request Body

```python
from pydantic import BaseModel
# Define schema for the request body
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.post("/purchase")
def read_items(item: Item):
    request_body = item.model_dump() # turn data into a Pyt
    return request_body
```

How to tell the Difference

PATH PARAMETER

- Appears inside the path wrapped in curly braces
- Is inside the function's arguments list

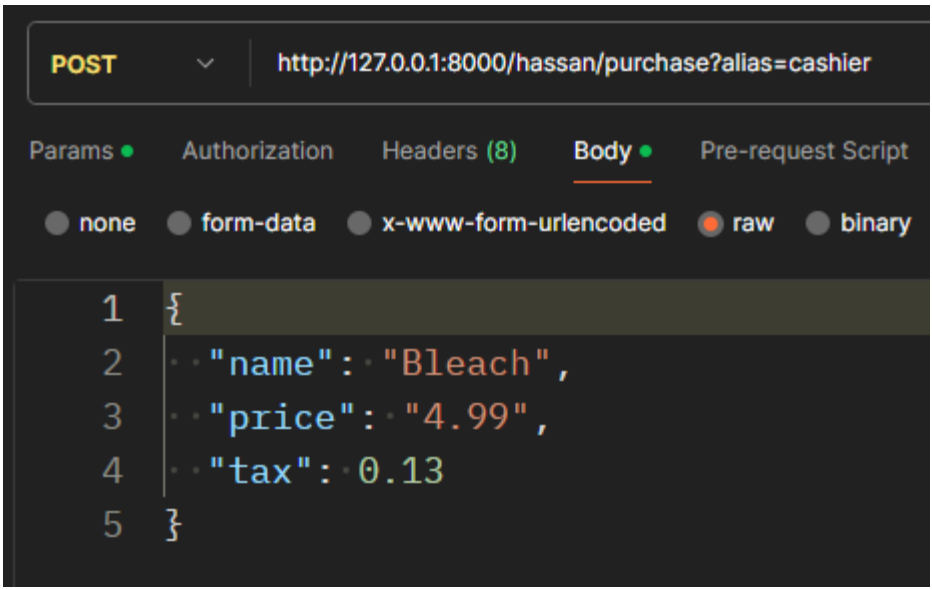
QUERY PARAMETER
Only appears inside the function's argument list

REQUEST BODY
Uses schema instead of a type definition

Ex. Path/Query Param + Request Body

Fast API is smart enough to tell which data is which
The order of the arguments in read_items is irrelevant

```python
from fastapi import FastAPI
from typing import Optional
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.post("/{username}/purchase")
def read_items(item: Item, username: str, alias: str): # th
    request_body = item.model_dump()
    return { "request_body": request_body, "path_param": us
```



```
{
    "request_body": {
        "name": "Bleach",
        "price": 4.99,
        "tax": 0.13
    },
    "path_param": "hassan",
    "query_param": "cashier"
}
```

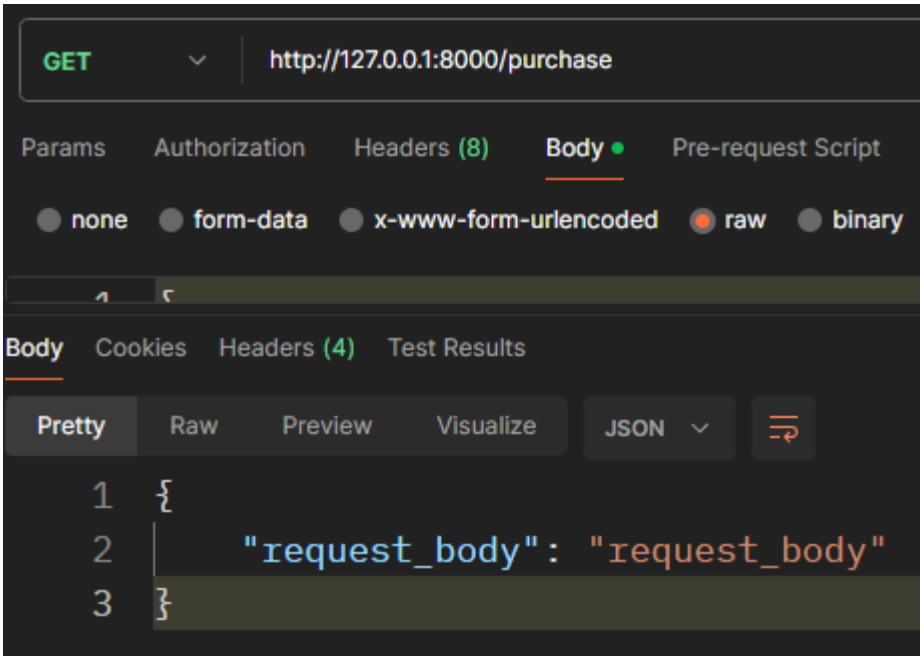
REST Endpoint Types

HTTP Requests Overview

GET: Fetch Data

Standard

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/purchase")
async def read_items():
    return {"request_body": "request_body"}
```

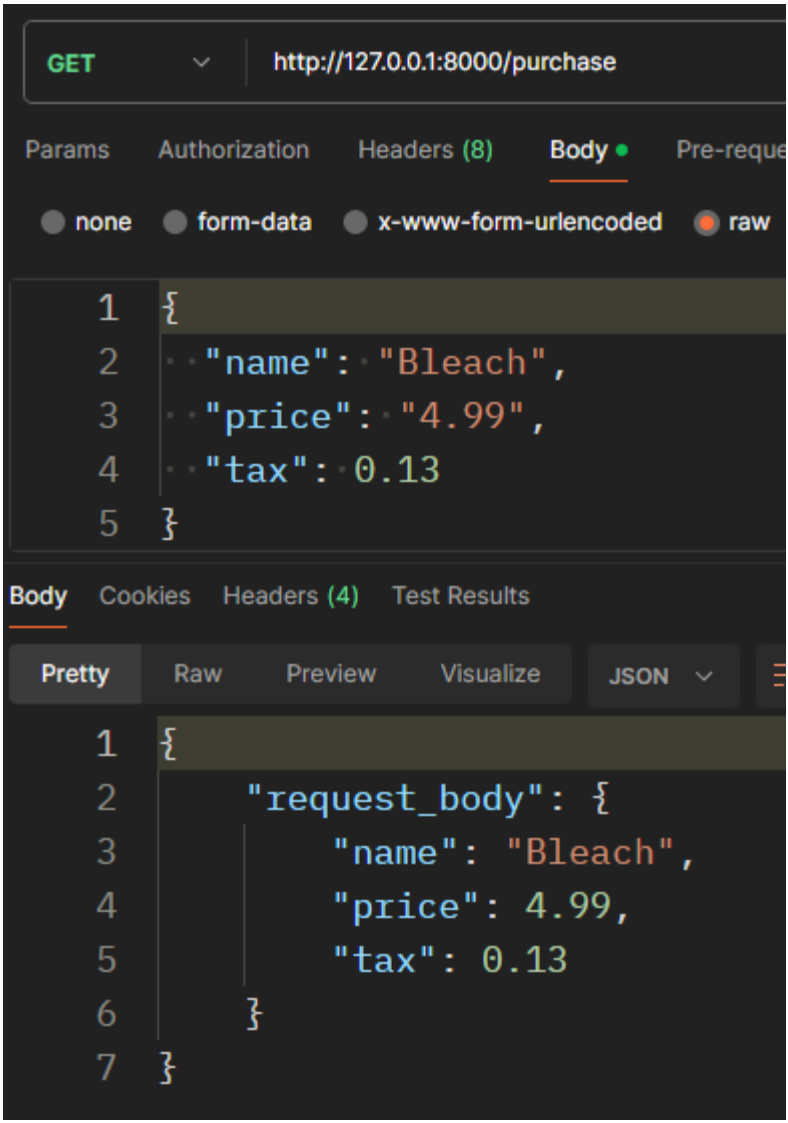


With Request Body

Although it's not typical, you actually can use a request body with GET requests

```
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.get("/purchase")
```

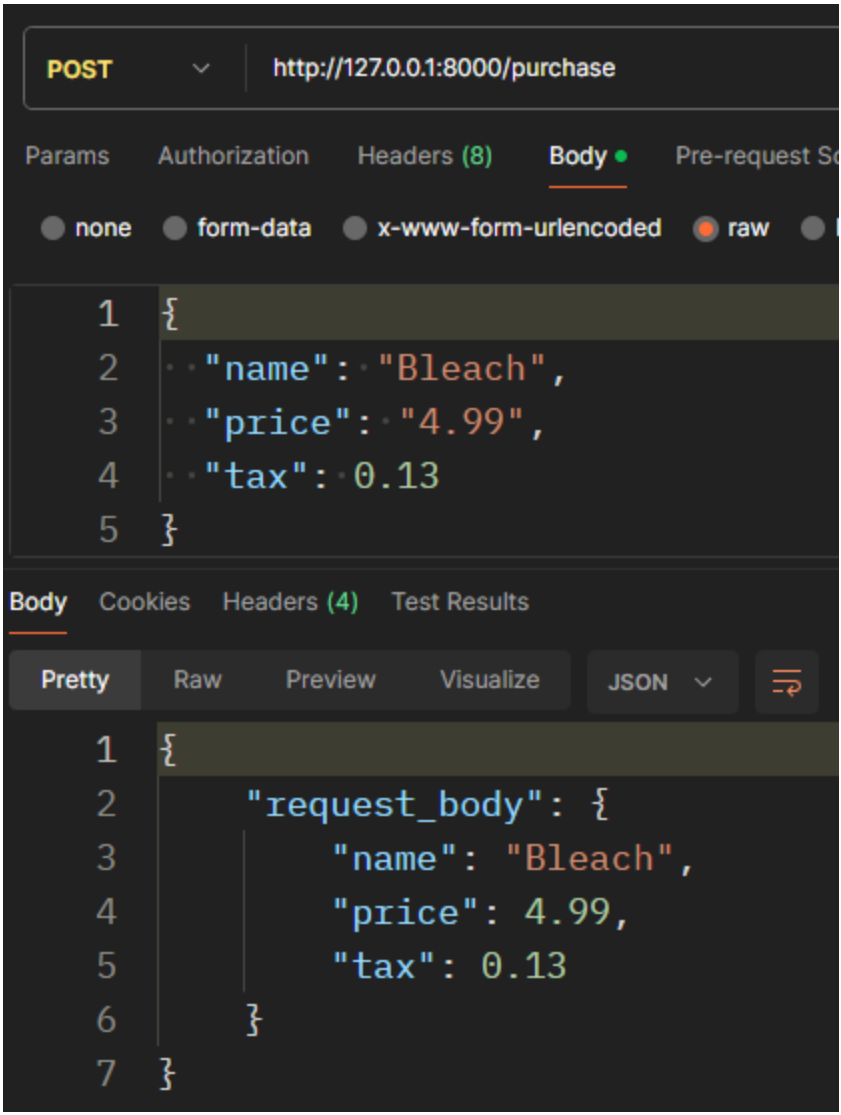
```python
async def read_items(item: Item):
    request_body = item.model_dump()
    return { "request_body": request_body }
```



POST: Upload new Data

Send a request body along with the POST request

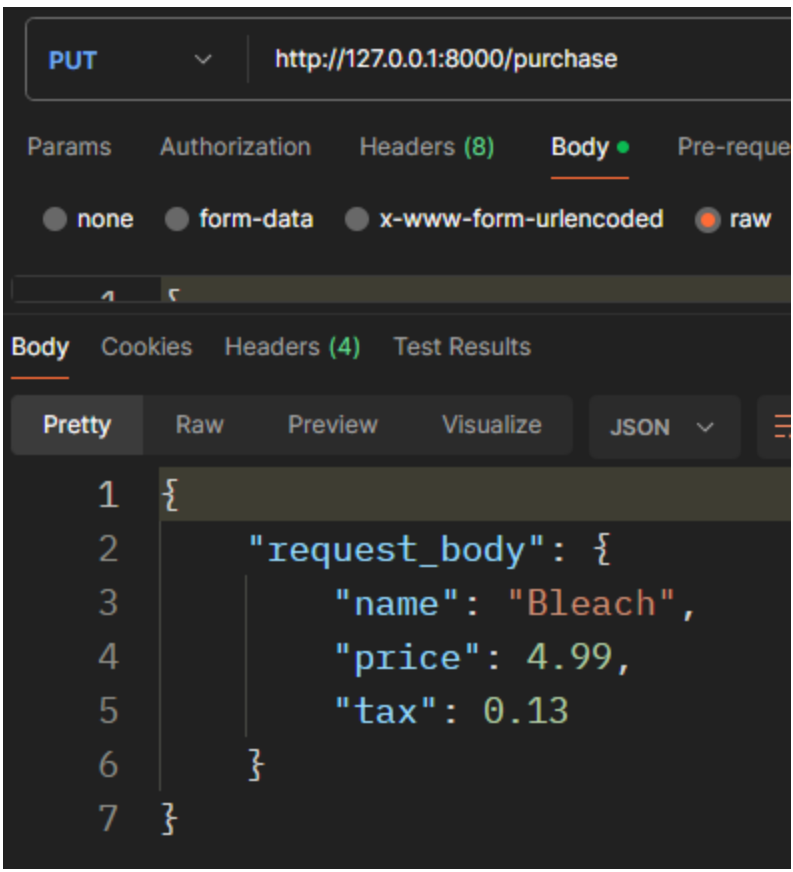
```python
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.post("/purchase")
async def read_items(item: Item):
    request_body = item.model_dump()
    return { "request_body": request_body }
```



PUT: Modify Entire Resource

Send a request body along with the PUT request

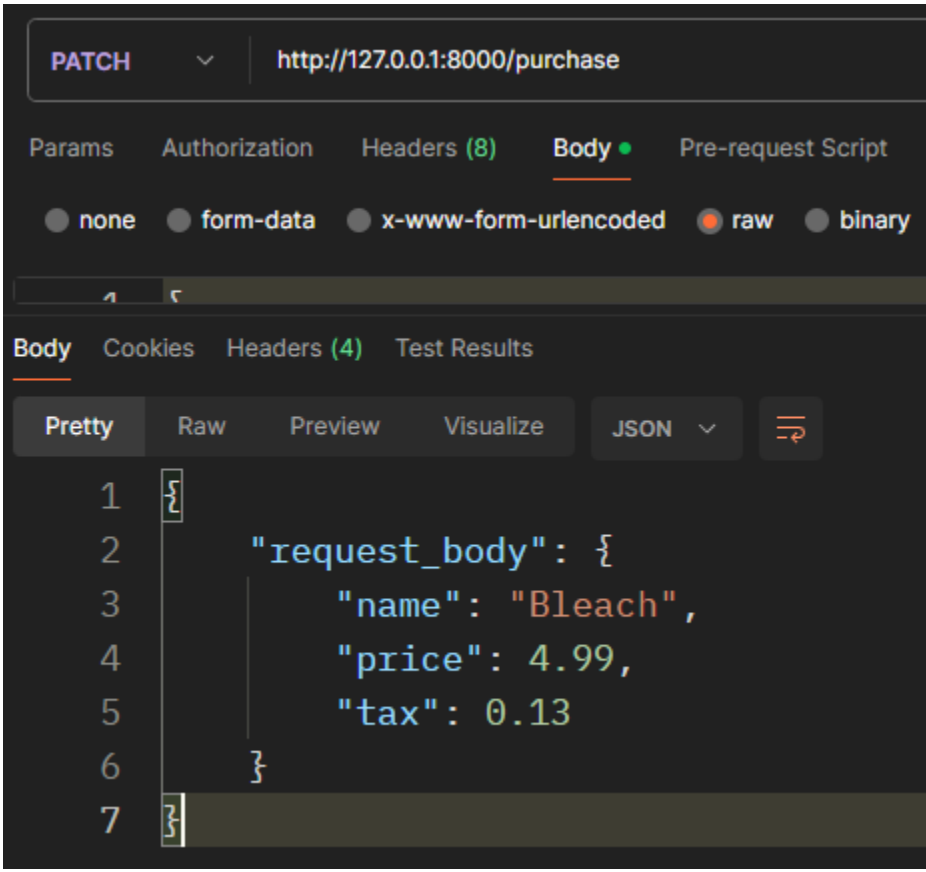
```python
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.put("/purchase")
async def read_items(item: Item):
    request_body = item.model_dump()
    return { "request_body": request_body }
```



PATCH: Partially Modify Resource

Send a request body along with the PATCH request

```python
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.patch("/purchase")
async def read_items(item: Item):
    request_body = item.model_dump()
    return { "request_body": request_body }
```

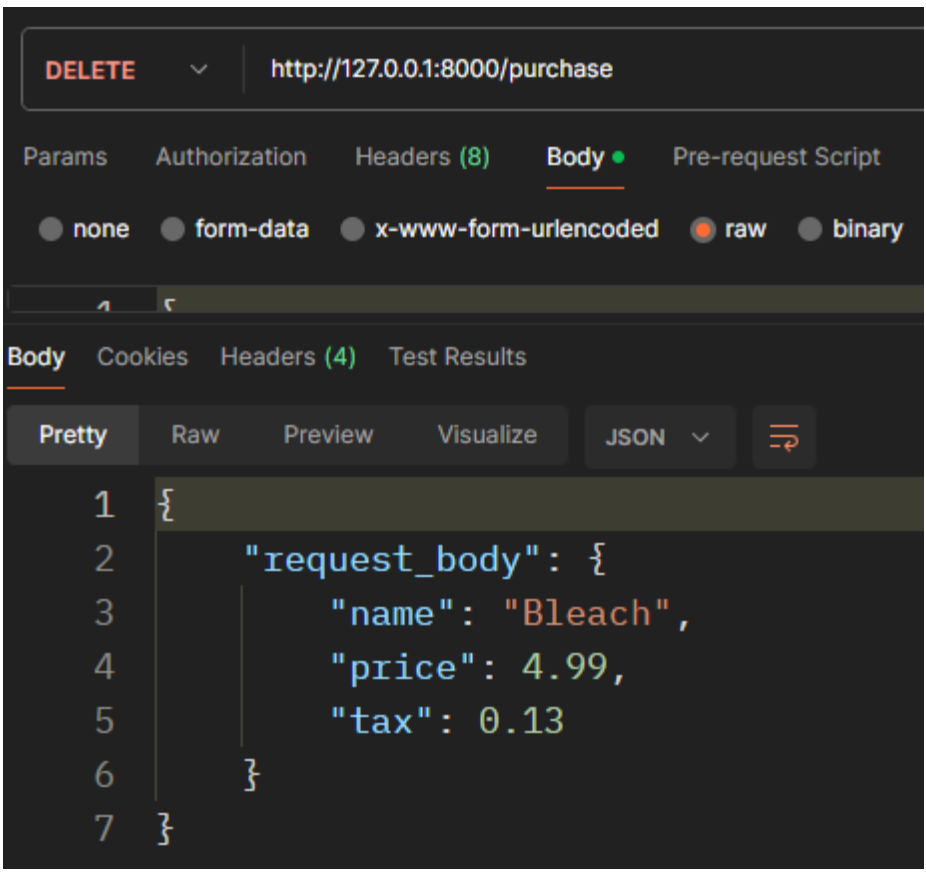


DELETE: Erase Data

Send a request body along with the DELETE request

```
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
```

```python
class Item(BaseModel):
    name: str  # Mandatory string
    price: float  # Mandatory float
    tax: float | None = None  # Optional float
@app.delete("/purchase")
async def read_items(item: Item):
    request_body = item.model_dump()
    return { "request_body": request_body }
```



Payload Type Validation >

Enums

You can specify a list of allowed values

```python
from pydantic import BaseModel
from enum import Enum
class FruitType(Enum):
    APPLE = "apple"
    ORANGE = "orange"
# Request Body Schema
class Item(BaseModel):
    name: str  # Mandatory string
    price: float | None # Optional float
    fruit: FruitType

@app.post("/test")
async def read_items(item: Item):
    request_body = item.model_dump()
    return { "request_body": request_body }
```

Request 1: Abiding by schema

Request 2: Ignoring Schema (error produced)

API Calls + Routing

API Calls w/ Requests Library

API call inside Endpoint

The requests library makes API calls in a synchronous manner - it halts code execution and you cannot run them concurrently
Do not need to `await` it, and an exception gets thrown if you try

```
from fastapi import FastAPI
import requests
app = FastAPI()
@app.get("/purchase")
async def read_items():
    data = requests.get("https://randomuser.me/api")
    parsed_data = data.json()
    return {"data": parsed_data}
```

API call Inside a Function

Must await the function making an endpoint call with async

```
from fastapi import FastAPI
import requests
app = FastAPI()
@app.get("/purchase")
async def read_items():
    data = await fetch_data()
    return {"data": data}
# Async function
async def fetch_data():
    payload = requests.get("https://randomuser.me/api")
    parsed = payload.json()
    return parsed
```

Routing

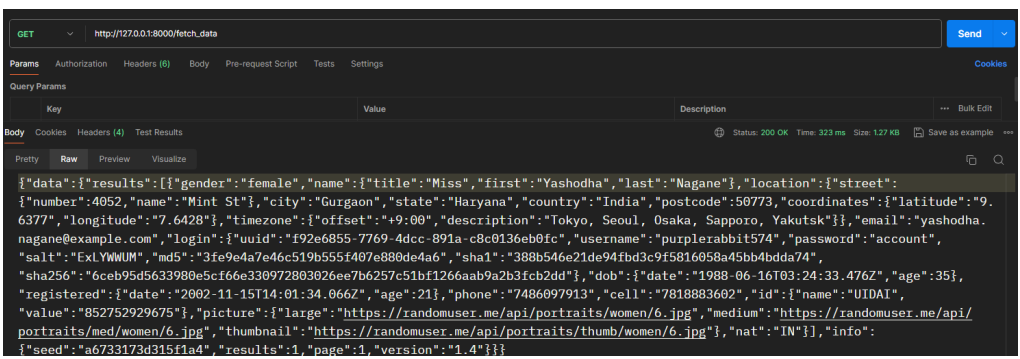
main.py (central file)

```
from fastapi import FastAPI
from endpoints.fetch_data import fetch_data_router
app = FastAPI()
app.include_router(fetch_data_router)
```

File holding endpoint

```python
from fastapi import APIRouter
import requests
fetch_data_router = APIRouter()
@fetch_data_router.get("/fetch_data")
async def endpoint():
    data = requests.get("https://randomuser.me/api")
    parsed_data = data.json()
    return {"data": parsed_data}
```

Make a GET request to `/fetch_data`



Project Initialization

Fetch Data on Startup and Share with All Endpoints

High Level Overview

Oftentimes you'll want all endpoints in a project to have access to certain data that can only be obtained from outside the project
The data could be in a database, 3rd party service...etc

1. Create a class that makes an async call and saves the data inside that class as a property

2. Make that async call inside server.py (or whatever your core Python file is) within a special lifespan method
   Feed that to the project initialization to ensure this call is made before the project spins up

3. Use dependency injection to provide this data to all your endpoints

Syntax

startup_data.py

```
import httpx
class StartupDataService:
    def __init__(self):
        self.TRANSACTION_TYPES = {}
        self.EMITTER_CATEGORIES = {}
        self.TRANSACTION_STATUSES = {}
    async def initialize(self):
        async with httpx.AsyncClient() as client:
            response = await client.get("https://jsonplaceh
            self.TRANSACTION_TYPES = response.json()
    def return_initial_data(self):
        return {
            "TRANSACTION_TYPES": self.TRANSACTION_TYPES,
        }
startup_data_store = StartupDataService() # used on project
def get_startup_data():
    return startup_data_store # needed for dependency injec
```

server.py

```
from fastapi import FastAPI
from contextlib import asynccontextmanager
from services.startup_data import startup_data_store
@asynccontextmanager
async def lifespan(app: FastAPI):
    # Fetch data before application starts
    await startup_data_store.initialize()
```

```
    yield
    # Cleanup: you could add cleanup code here if needed
app = FastAPI(lifespan=lifespan)
```

some_endpoint.py

```
from fastapi import APIRouter, HTTPException, Depends
from optchain_next_backend_service.services.startup_data im
router = APIRouter()
@router.post("/create_transaction")
async def create_transaction(
    startup_data: StartupDataService = Depends(get_startup_
):
    print("99Z", startup_data.TRANSACTION_TYPES)
    return CreateTransactionResponse(details="Change later"
```

Asyncio

What Asyncio does

Asyncio is used behind the scenes in FastAPI to enable async programming
Every time you declare an async function - asyncio is running in the background

```
async def create_events():
    results = await fetch_data()
```

asyncio.sleep() : Artificial Delay

Creates awaitable artificial delays

```
async def fake_fetch_data():
    await asyncio.sleep(5) # 5 second delay
    return "Fetched data"
```

asyncio.gather() : Run Async Operations in Parallel

Works like Promise.all in JS

```python
import asyncio
import random
async def fetch_data(item_id):
    """Simulates fetching data."""
    await asyncio.sleep(random.uniform(0.5, 2))
    return f"Data for item {item_id}"
async def main():
    tasks = [
        fetch_data(1),
        fetch_data(2),
        fetch_data(3),
        fetch_data(4)
    ]
    results = await asyncio.gather(*tasks) # unpack the lis
    print("\nAll results:")
    for result in results:
        print(result)
```

< Middleware >

https://fastapi.tiangolo.com/tutorial/middleware/

Google Cloud

API Table of Contents
https://cloud.google.com/storage/docs

Buckets Overview
https://cloud.google.com/storage/docs/buckets

Upload objects in memory to a bucket
https://cloud.google.com/storage/docs/uploading-objects-from-memory